(AS)2 : accelerator synthesis using algorithmic skeletons for rapid design space exploration
Fernando, S.D.; Wijtvliet, M.; Nugteren, C.; Kumar, A.; Corporaal, H.

Published in:
Proceedings of the Conference on Design, Automation and Test in Europe (DATE 15) 9-13 March 2015, Grenoble, France

Published: 01/01/2015

Document Version
Accepted manuscript including changes made at the peer-review stage

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):
Abstract—Hardware accelerators in heterogeneous multiproces-
sor system-on-chips are becoming popular as a means of
meeting performance and energy efficiency requirements of mod-
erm embedded systems. Current design methods for accelerator
synthesis, such as High-Level Synthesis, are not fully automated.
Therefore, time consuming manual iterations are required to
explore efficient accelerator alternatives: the programmer is still
required to think in terms of the underlying architecture. In
this paper, we present (AS)$^2$: a design flow for Accelerator
Synthesis using Algorithmic Skeletons. Skeletonization separates
the structure of a parallel computation from an algorithms’ func-
tionality, enabling efficient implementations without requiring the
programmer to have hardware knowledge. We define three such
skeletons (for three image processing kernels) enabling FPGA
specific parallelization techniques and optimizations. As a case
study, we present a design space exploration of these skeletons
and show how multiple design points with area–performance
trade-offs for the accelerators can be efficiently and rapidly
synthesized. We show that (AS)$^2$ is a promising direction for
accelerator synthesis as it generates a pareto front of 8 design
points in under half an hour for each of the three image
processing kernels.

I. INTRODUCTION

Hardware accelerators in popular heterogeneous multipro-
cessor system-on-chip platforms (e.g. Xilinx Zynq) combine
FPGA fabric with general purpose processors, and provide
performance gains and higher energy efficiency [1], compared
to a software implementation. However, manual accelerator
design requires writing Register-Transfer Level (RTL) code,
which is error-prone and time consuming. High-Level Synthe-
sis (HLS) is a promising solution for this accelerator synthesis
problem. However, the programmer has to restructure the code,
manually and iteratively, to obtain an efficient implementation
[2, 3]. Hence, exploring multiple accelerator trade-offs is non-
trivial and the key challenge is to rapidly explore them.

A promising solution for this challenge is using algorithmic
skeletons. The main advantage of using skeletons is that
the knowledge of the hardware target can be captured in a
skeleton template to generate efficient accelerators. By using
algorithmic skeletons, a kernel in C code can be classified
into a group with a characteristic memory access pattern -
an algorithmic species - and a skeleton template is chosen
for target code generation. A skeleton template describes an
optimized, parameterized implementation for a species. Algo-

(AS)$^2$: Accelerator Synthesis using Algorithmic Skeletons
for Rapid Design Space Exploration

Shakith Fernando$^1$, Mark Wijtvliet$^1$, Cedric Nugteren$^1$, Akash Kumar$^2$ and Henk Corporaal$^1$

$^1$Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands
$^2$Department of Electrical & Computer Engineering, National University of Singapore, Singapore

Corresponding author email: s.fernando@tue.nl

The following are the key contributions of this paper:

- A novel parametric algorithmic skeleton based design
  flow for FPGA accelerator synthesis.
- Three example parametric skeletons, enabling FPGA spe-
cific parallelization techniques and optimizations.
- A case study using these skeletons for three image
  processing kernels to show rapid design space exploration
  for accelerator synthesis.
- The comparison of area–performance trade-offs of accel-
erators generated with our design flow compared to accel-
erators generated with naive C (both through HLS), man-
ually generated hand-coded RTL, and generated through
the Vivado OpenCV video library [9].
II. Survey of Related Work

Methods for accelerator synthesis are widely studied in literature. They are listed and compared in Table I.

Using high level structural languages, a programmer directly writes the hardware structure to generate the RTL. SystemC, JHDL and MyHDL are such examples. This approach can generate highly efficient and parallel hardware structures, depending on how the programmer writes the structural code. However, this also means that the programmer has to have knowledge of hardware design and perform the optimization steps (e.g. extracting parallelism) manually.

Compilation based approaches (HLS) take high level descriptions (e.g. C code) as input and generate parallel hardware structures. Typically, user-defined directives guide the parallel hardware structure generation[8]. Although HLS can generate efficient accelerators, it requires the programmer to manually iterate using directives, code transformations and vendor specific coding styles [5, 10]. These iterations are required due to the black box nature of HLS [2].

Due to the drawbacks of HLS mentioned above, there is a recent emergence of C-based accelerator libraries for HLS [9], similar to how accelerator synthesis was performed, traditionally, using hand-coded RTL libraries. The required vendor specific coding styles, directives and code transformations are pre-built into function libraries for efficient accelerator generation. However, the number of functions in these domain specific libraries is limited and they only support a single design point.

Skeleton-based parallelization [11] for FPGA accelerator synthesis is first reported by HIDE [6]. This is a method that uses image algebra based system descriptions as input and automatically generates accelerators. It is similar to (AS)² with RTL as target code. However, the user must provide the structural representation, which limits the rapid exploration of multiple design points. Furthermore, the functional operations are limited to the HIDE arithmetic IP library.

Recently, Matai et al. [3] and Schmid et al. [7] have proposed skeleton-based methods for accelerator synthesis. Matai et al. [3] have only proposed the idea of template libraries to solve the challenges of accelerator synthesis and they don’t provide any details nor any implementations. Schmid et al. [7] is similar to our work; however, the programmer must classify the code and select the correct template. Furthermore, based on their results, they currently support only a template for a single pattern and they are limited to a single design point for each kernel.

In contrast to the discussed research works, (AS)² is unique in several aspects:

- It takes in sequential C code, so that the programmer can abstract from the hardware architecture.
- It allows for the generation of efficient accelerators by encoding FPGA parallelization techniques per skeleton.
- It allows for the rapid generation of multiple design points for accelerators through parametric skeleton templates.

III. Algorithmic Species

We illustrate algorithmic species by showing example access patterns of an image processing kernel: erosion (Listing 1). To produce a single element of the result array B, it needs a 3x3 neighbourhood from input array A (Figure 2a). A and B have access patterns of neighbourhood and element respectively and form a single species (Table II) when combined.

```c
Listing 1. Erosion

for( i = 0; i < h; i++)  // h is 4
for( j = 0; j < w; j++)  // w is 8
if ( (i < 1) || (i>=h-1) || (j < 1) || (j>=w-1) )
  B[i][j]= 0;
else
  && A[i+1][j-1] && A[i+1][j] && A[i+1][j+1]);
```

Fig. 2. Illustration of the first two iterations of the three image processing kernels
Table II further classifies two other kernels (Listings 2 and 3) into algorithmic species. In Listing 2, to produce a single element of the output array \( B \), it needs a single element from input array \( A \) (Figure 2b). In Listing 3, to produce a single element of the output array \( B \), it needs a \( w \) sized chunk of data (\( w \) is the width of the input data frame) from input array \( A \) (Figure 2c). The neighbourhood and chunk patterns differ in the memory accesses that overlap across loop iterations. Then, for each kernel with a known algorithmic species, an accelerator can be synthesised using a template. These patterns can be derived manually or automatically using A-DARWIN [4]. It should be noted that species with similar characteristics can be mapped onto the same parameterizable template.

IV. DESIGN FLOW

In this section, we present details of our design flow. Figure 4 shows two inputs: (1) C code annotated with species, and (2) user defined parameters. A template based on the species is selected and instantiated with the functional operations extracted from the input C code to generate the optimized target code.

The parameters allow the user to select from multiple skeleton instantiations for each species. \( P_0 \) defines the spatial parallelism factor. \( P_0 = 1 \) instantiates a skeleton template that is optimized for a single operation, whereas \( P_0 = w \) instantiates a skeleton template that is optimized for a parallel line-based operation. \( P_1 \) selects different vendor specific HLS coding styles, while \( P_2 \) selects the target code (C’ or RTL). The optimize step can contain several compiler passes.

We illustrate our design flow using the projection kernel (Listing 3). The C code is then annotated with a species classification we presented in Table II. The resulting annotated C code is used as the input for our design flow, which uses the species classification in combination with the user specified parameters (specifying parallelism and target code) to generate the C’ code shown in Listing 4. The generated C’ code is the input for the HLS tool. The C’ code contains HLS directives to guide the HLS tool to obtain a more efficient implementation. The listing of C’ code also contains code transformations to help the HLS tool to recognize parallelism.

```
1 void projection(In_t A[w*h], Out_t B[h])
2 #pragma HLS INLINE off
3 for (i=0; i<h; i++)
4 COMPUTATION:for (j=0; j<w; j=j+2)
5 #pragma HLS PIPELINE II=1
6 
7 COMPUTATION:for (j = 0; j < w; j=j+2)\{
8 #pragma HLS EXPRESSION_BALANCE
9 #pragma HLS UNROLL
10 sum_temp = sum_temp + A[w*i+j+p];\}
11 B[i] = sum_temp;
```

A more complex example of the generated accelerator structure for erosion (Listing 1) is shown in Figure 3. The accelerator has 4 processing stages (Figure 3b). Stage 0 loads data from the input array \( A \) to a window buffer. Stage 1a directly loads the buffered data to the computation datapath in stage 2. In stage 3, the output elements are written to array \( B \). To exploit reuse, the data in the window buffer (stage 1b) is shifted left and the next two columns of data are loaded. Processing continues on the next row until the complete output frame is generated.
In this section, we present the results of the design space exploration for erosion, binarization and projection kernels (Listings 1, 2 and 3) with $w = 128$ and $h = 32$ ($h$ is the height of the frame). These results are compared against several surveyed methods in Section II. Vivado 2013.1 was used for HLS, and Xilinx ISE 14.4 was used for logic synthesis. The implementation target was the Zynq XC7Z045 FPGA device.

Figure 5 shows the area–performance trade-offs for the three kernels using: (1) a single design point from a hand-coded HLS implementation, and Xilinx ISE 14.4 was used for logic synthesis. The implementation target was the Zynq XC7Z045 FPGA device.

The drawback of HLS is illustrated in Figure 5a. This shows that using HLS on naive sequential code (HLS-C) gives worse performance when it is parallelized. This is because the HLS-C implementations are limited by the number of block RAM ports, whereas the (AS)$^2$ implementations make use of memory access restructuring described in Section IV. In contrast, the design points generated by (AS)$^2$ (except for $P_0 = 2$) form a pareto front. The actual deviation in $P_0 = 4$ was due to logic synthesis packing the buffers more efficiently, thus reducing slice usage. The library design point is dominated by (AS)$^2$ for the $P_0 = 1$ case. The library point is a generic implementation to support multiple filter mask types, whereas the implementation generated by (AS)$^2$ is optimized for the erosion operation with a rectangular mask. The difference between the hand-coded implementation and the $P_0 = 128$ case is because the hand-coded RTL implementation needs 4 cycles to load 4x32-bit elements for a single line of 128 bits, whereas $P_0 = 128$ loads the complete 128-bit line in a single cycle as it has 128 banks. Though this leads to a performance gain, it also leads to an area overhead in combination with overheads introduced by HLS.

Figure 5b presents the results of binarization, an embarassingly parallel algorithm. In this case, HLS and (AS)$^2$ show similar performance.

In Projection (see Figure 5c), the (AS)$^2$ implementations dominate HLS-C implementations, as the skeleton template contains code transformations to enable balanced operator tree detection and to enable pipelining in this tree (deep pipelining in the case of $P_0 = 128$). Finally, the difference between the hand-coded implementation and the $P_0 = 32$ case is because the hand-coded RTL implementation is not pipelined.

### V. EXPERIMENTS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Resource Usage [Slices]</th>
<th>Execution Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library Erosion</td>
<td>Library Binarization</td>
<td>Library Projection</td>
</tr>
<tr>
<td>(AS)$^2$ Erosion</td>
<td>(AS)$^2$ Binarization</td>
<td>(AS)$^2$ Projection</td>
</tr>
<tr>
<td>HLS-C Erosion</td>
<td>HLS-C Binarization</td>
<td>HLS-C Projection</td>
</tr>
</tbody>
</table>

Fig. 5. Design Space Exploration Results

### VI. CONCLUSION

In this paper, we proposed a design flow for accelerator synthesis using algorithmic skeletons. The skeletonization separates the structure of the parallel computation from the algorithm’s functionality, enabling efficient FPGA implementations. A case study is presented to find area–performance trade-offs for three species. It shows how multiple efficient design points for accelerators can be rapidly synthesized.

Currently, the skeleton library is limited to C’ target code. We plan to extend that with RTL skeleton templates and to fully automate the tool flow. To show scalability of the design flow, we would like to extend the results with more benchmark kernels for each species. Furthermore, we would like to model algorithmic species to predict performance, area and power usage at an earlier stage of the design flow.

### REFERENCES


